DOT Language

Abstract grammar for defining Graphviz nodes, edges, graphs, subgraphs, and clusters.

Terminals are shown in bold font and nonterminals in italics. Literal characters are given in single quotes. Parentheses (and) indicate grouping when needed. Square brackets [and] enclose optional items. Vertical bars | separate alternatives.

graph	:	[**strict**] (**graph** | **digraph**) [*ID*] **'{'** *stmt_list* **'}'**
stmt_list	:	[*stmt* [**';'**] *stmt_list*]
stmt	:	*node_stmt*
	|	*edge_stmt*
	|	*attr_stmt*
	|	*ID* **'='** *ID*
	|	*subgraph*
attr_stmt	:	(**graph** | **node** | **edge**) *attr_list*
attr_list	:	**'['** [*a_list*] **']'** [*attr_list*]
a_list	:	*ID* **'='** *ID* [(**';'** | **','**)] [*a_list*]
edge_stmt	:	(*node_id* | *subgraph*) *edgeRHS* [*attr_list*]
edgeRHS	:	*edgeop* (*node_id* | *subgraph*) [*edgeRHS*]
node_stmt	:	*node_id* [*attr_list*]
node_id	:	*ID* [*port*]
port	:	**':'** *ID* [**':'** *compass_pt*]
	|	**':'** *compass_pt*
subgraph	:	[**subgraph** [*ID*]] **'{'** *stmt_list* **'}'**
compass_pt	:	(**n** | **ne** | **e** | **se** | **s** | **sw** | **w** | **nw** | **c** | _)

The keywords **node**, **edge**, **graph**, **digraph**, **subgraph**, and **strict** are case-independent. Note also that the allowed compass point values are not keywords, so these strings can be used elsewhere as ordinary identifiers and, conversely, the parser will actually accept any identifier.

An *ID* is one of the following:

- Any string of alphabetic (`[a-zA-Z\200-\377]`) characters, underscores (`'_'`) or digits(`[0-9]`), not beginning with a digit;
- a numeral $[\ -\ ]^{?}(\ .\ [\ 0\ -\ 9\ ]^{+}\ \ |\ \ [\ 0\ -\ 9\ ]^{+}(\ .\ [\ 0\ -\ 9\ ]^{*})^{?}$);
- any double-quoted string (`"..."`) possibly containing escaped quotes (`\"`)[1];
- an HTML string (`<...>`).

An ID is just a string; the lack of quote characters in the first two forms is just for simplicity. There is no semantic difference between `abc_2` and `"abc_2"`, or between `2.34` and `"2.34"`. Obviously, to use a keyword as an ID, it must be quoted. Note that, in HTML strings, angle brackets must occur in matched pairs, and newlines and other formatting whitespace characters are allowed. In addition, the content must be legal XML, so that the special XML escape sequences for ", &, <, and > may be necessary in order to embed these characters in attribute values or raw text. As an ID, an HTML string can be any legal XML string. However, if used as a label attribute, it is interpreted specially and must follow the syntax for HTML-like labels.

Both quoted strings and HTML strings are scanned as a unit, so any embedded comments will be treated as part of the strings.

An *edgeop* is `->` in directed graphs and `--` in undirected graphs.

The language supports C++-style comments: `/* */` and `//`. In addition, a line beginning with a '#' character is considered a line output from a C preprocessor (e.g., # 34 to indicate line 34) and discarded.

Semicolons and commas aid readability but are not required. Also, any amount of whitespace may be inserted between terminals.

As another aid for readability, dot allows double-quoted strings to span multiple physical lines using the standard C convention of a backslash immediately preceding a newline character[2]. In addition, double-quoted strings can be concatenated using a '+' operator. As HTML strings can contain newline characters, which are used solely for formatting, the language does not allow escaped newlines or concatenation operators to be used within them.

Subgraphs and Clusters

Subgraphs play three roles in Graphviz. First, a subgraph can be used to represent graph structure, indicating that certain nodes and edges should be grouped together. This is the usual role for subgraphs and typically specifies semantic information about the graph components. It can also provide a convenient shorthand for edges. An edge statement allows a subgraph on both the left and right sides of the edge operator. When this occurs, an edge is created from every node on the left to every node on the right. For example, the specification

```
A -> {B C}
```

is equivalent to

```
A -> B
A -> C
```

In the second role, a subgraph can provide a context for setting attributes. For example, a subgraph could specify that blue is the default color for all nodes defined in it. In the context of graph drawing, a more interesting example is:

```
subgraph {
  rank = same; A; B; C;
}
```

This (anonymous) subgraph specifies that the nodes A, B and C should all be placed on the same rank if drawn using dot.

The third role for subgraphs directly involves how the graph will be laid out by certain layout engines. If the name of the subgraph begins with `cluster`, Graphviz notes the subgraph as a special *cluster* subgraph. If supported, the layout engine will do the layout so that the nodes belonging to the cluster are drawn together, with the entire drawing of the cluster contained within a bounding rectangle. Note that, for good and bad, cluster subgraphs are not part of the DOT language, but solely a syntactic convention adhered to by certain of the layout engines.

Lexical and Semantic Notes

A graph must be specified as either a **digraph** or a **graph**. Semantically, this indicates whether or not there is a natural direction from one of the edge's nodes to the other. Lexically, a digraph must specify an edge using the edge operator `->` while a undirected graph must use `--`. Operationally, the distinction is used to define different default rendering attributes. For example, edges in a digraph will be drawn, by default, with an arrowhead pointing to the head node. For ordinary graphs, edges are drawn without any arrowheads by default.

A graph may also be described as **strict**. This forbids the creation of multi-edges, i.e., there can be at most one edge with a given tail node and head node in the directed case. For undirected graphs, there can be at most one edge connected to the same two nodes. Subsequent edge statements using the same two nodes will identify the edge with the previously defined one and apply any attributes given in the edge statement. For example, the graph

```
strict graph {
  a -- b
  a -- b
  b -- a [color=blue]
}
```

will have a single edge connecting nodes `a` and `b`, whose color is blue.

If a default attribute is defined using a **node**, **edge**, or **graph** statement, or by an attribute assignment not attached to a node or edge, any object of the appropriate type defined afterwards will inherit this attribute value. This holds until the default attribute is set to a new value, from which point the new value is used. Objects defined before a default attribute is set will have an empty string value attached to the attribute once the default attribute definition is made.

Note, in particular, that a subgraph receives the attribute settings of its parent graph at the time of its definition. This can be useful; for example, one can assign a font to the root graph and all subgraphs will also use the font. For some attributes, however, this property is undesirable. If one attaches a label to the root graph, it is probably not the desired effect to have the label used by all subgraphs. Rather than listing the graph attribute at the top of the graph, and the resetting the attribute as needed in the subgraphs, one can simply defer the attribute definition in the graph until the appropriate subgraphs have been defined.

If an edge belongs to a cluster, its endpoints belong to that cluster. Thus, where you put an edge can effect a layout, as clusters are sometimes laid out recursively.

There are certain restrictions on subgraphs and clusters. First, at present, the names of a graph and it subgraphs share the same namespace. Thus, each subgraph must have a unique name. Second, although nodes can belong to any number of subgraphs, it is assumed clusters form a strict hierarchy when viewed as subsets of nodes and edges.

Character encodings

The DOT language assumes at least the ASCII character set. Quoted strings, both ordinary and HTML-like, may contain non-ASCII characters. In most cases, these strings are uninterpreted: they simply serve as unique identifiers or values passed through untouched. Labels, however, are meant to be displayed, which requires that the software be able to compute the size of the text and determine the appropriate glyphs. For this, it needs to know what character encoding is used.

By default, DOT assumes the UTF-8 character encoding. It also accepts the Latin1 (ISO-8859-1) character set, assuming the input graph uses the **charset** attribute to specify this. For graphs using other character sets, there are usually programs, such as `iconv`, which will translate from one character set to another.

Another way to avoid non-ASCII characters in labels is to use HTML entities for special characters. During label evaluation, these entities are translated into the underlying character. This table shows the supported entities, with their Unicode value, a typical glyph, and the HTML

entity name. Thus, to include a lower-case Greek beta into a string, one can use the ASCII sequence `&beta;`. In general, one should only use entities that are allowed in the output character set, and for which there is a glyph in the font.

---

1. In quoted strings in DOT, the only escaped character is double-quote `"`. That is, in quoted strings, the dyad `\"` is converted to `"`; all other characters are left unchanged. In particular, `\\` remains `\\`. Layout engines may apply additional escape sequences.
2. Previous to 2.30, the language allowed escaped newlines to be used anywhere outside of HTML strings. The new lex-based scanner makes this difficult to implement. Given the perceived lack of usefulness of this generality, we have restricted this feature to double-quoted strings, where it can actually be helpful.

Last modified October 4, 2022: description for Language (7acfe98)